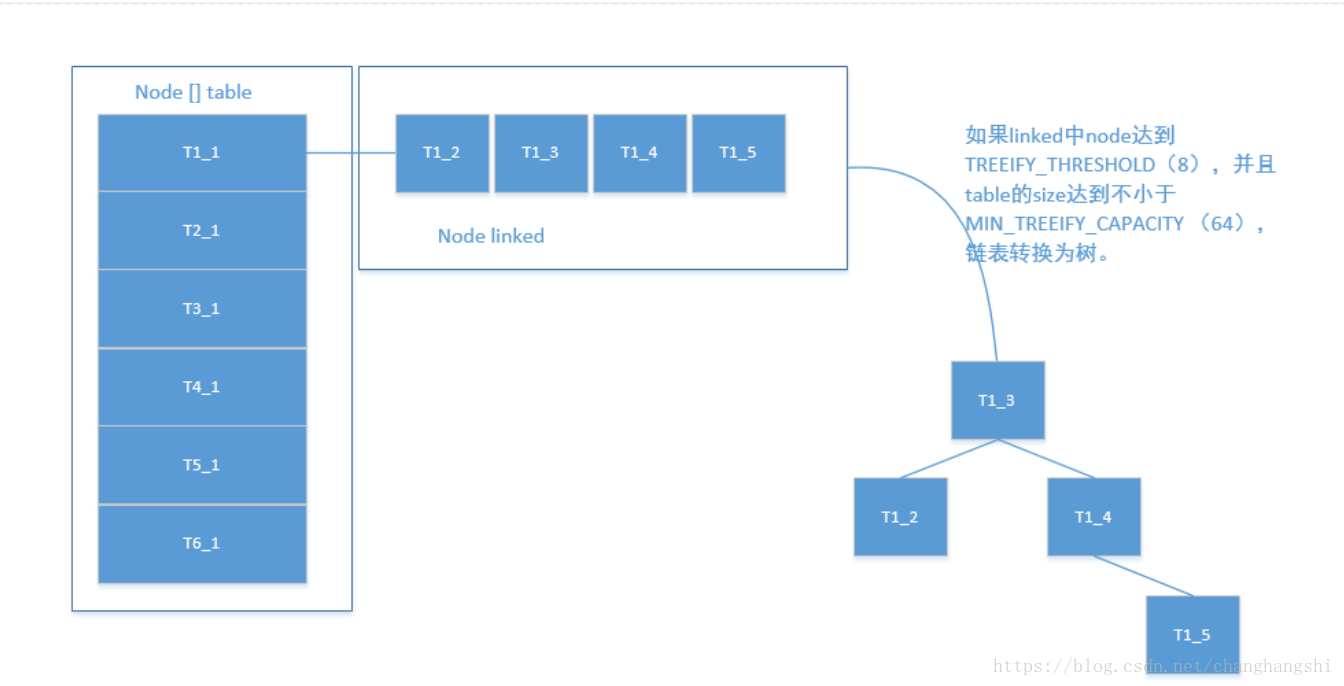
Jdk1.7 到 1.8 hashmap 发生了什么变化？

数据结构发生了变化，从之前单纯的数组+链表结构变成了数组+链表+红黑树，

也就是说在JVM存储hashmap的K-V时，仅仅通过key来决定每一个entry的存储槽位（Node[]中的index）。并且Value以链表的形式挂在对应的槽位上（1.8以后如果value长度大于8则转化为红黑树）。

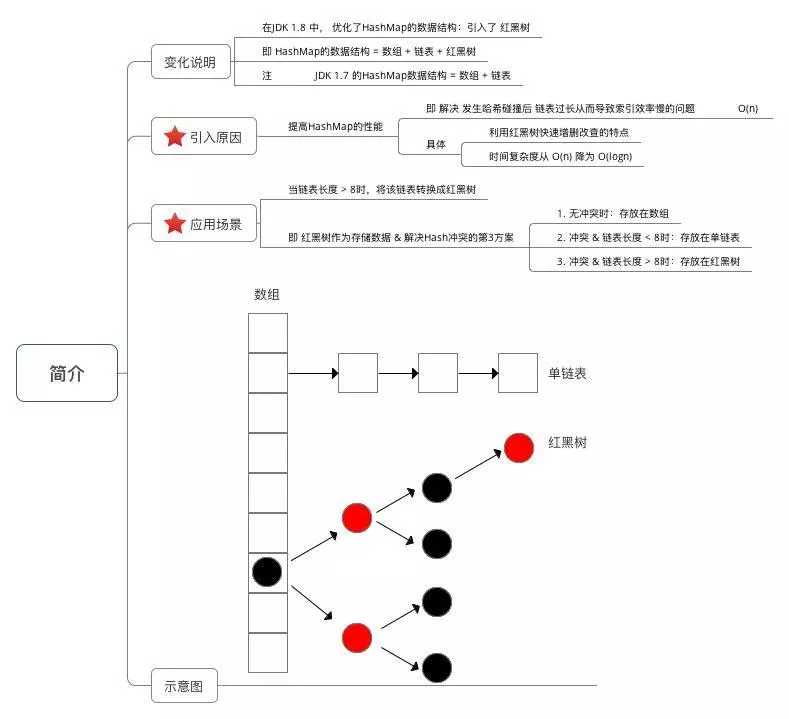


数据存入过程：

1.接受传入的参数，通过key计算hash值，得到数组下标位置；未发生hash碰撞，直接插入结束；发生hash碰撞，走第2步；

2.判断当前数据节点是红黑树还是链表，如果是链表，将数据放入链表头节点，原数据往后移；如果是红黑树，走第3步；

3.直接在红黑树插入数据结束；



但是hashmap1.7和1.8中都没有任何同步操作，容易出现并发问题，甚至出现死循环，导致系统不可用。

解决方案是jdk的ConcurrentHashMap, 位于java.util.concurrent下，专门解决并发问题

**详解**：<https://blog.csdn.net/changhangshi/article/details/82114727>

扩展：二叉树和红黑树的区别：

分布式事务的数据库实现MVCC:

MVCC是一种多版本并发控制机制

经典解析：<https://blog.csdn.net/SnailMann/article/details/94724197>

参考官网：<https://dev.mysql.com/doc/refman/8.0/en/innodb-multi-versioning.html>

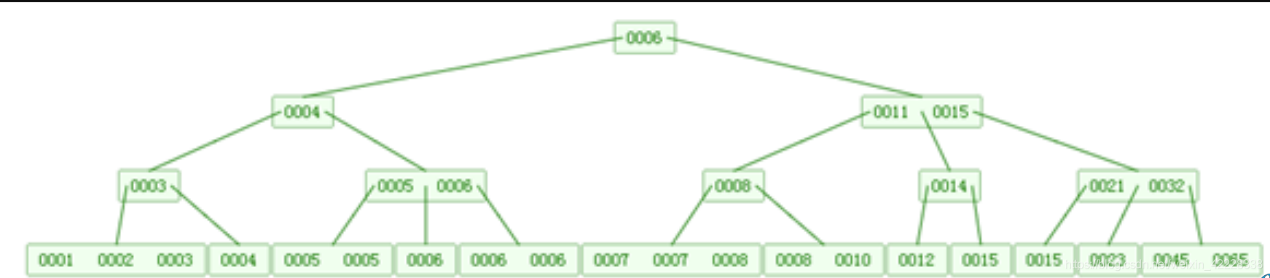
B树和B+树的区别

B树:

M阶B树的除根节点外的所有非叶节点的关键字取值区间为[M/2-1(向上取整),M-1] 并且以升序排列。

例子：依次插入6 10 4 14 5 11 15 3 2 12 1 7 8 8 6 3 6 21 5 15 15 6 32 23 45 65 7 8 6 5 4





B+树:

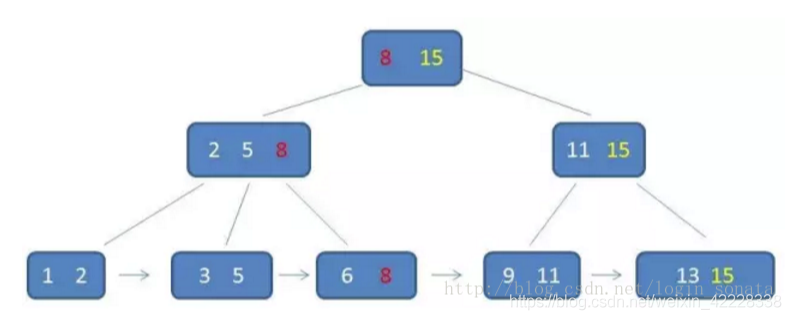
有n棵子树的非叶子结点中含有n个关键字（b树是n-1个），这些关键字不保存数据，只用来索引，所有数据都保存在叶子节点（b树是每个关键字都保存数据）。

所有的叶子结点中包含了全部关键字的信息，及指向含这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大顺序链接（叶子节点组成一个链表）。

所有的非叶子结点可以看成是索引部分，结点中仅含其子树中的最大（或最小）关键字。

通常在b+树上有两个头指针，一个指向根结点，一个指向关键字最小的叶子结点。

同一个数字会在不同节点中重复出现，根节点的最大元素就是b+树的最大元素。



B树与B+树的区别

B树每个节点都存储数据，所有节点组成这棵树。B+树只有叶子节点存储数据（B+数中有两个头指针：一个指向根节点，另一个指向关键字最小的叶节点），叶子节点包含了这棵树的所有数据，所有的叶子结点使用链表相连，便于区间查找和遍历，所有非叶节点起到索引作用。

B树中叶节点包含的关键字和其他节点包含的关键字是不重复的，B+树的索引项只包含对应子树的最大关键字和指向该子树的指针，不含有该关键字对应记录的存储地址。

B树中每个节点（非根节点）关键字个数的范围为[m/2(向上取整)-1,m-1](根节点为[1,m-1])，并且具有n个关键字的节点包含（n+1）棵子树。B+树中每个节点（非根节点）关键字个数的范围为[m/2(向上取整),m](根节点为[1,m])，具有n个关键字的节点包含（n）棵子树。

B+树中查找，无论查找是否成功，每次都是一条从根节点到叶节点的路径。

B树的优点

B树的每一个节点都包含key和value，因此经常访问的元素可能离根节点更近，因此访问也更迅速。

B+树的优点

所有的叶子结点使用链表相连，便于区间查找和遍历。B树则需要进行每一层的递归遍历。相邻的元素可能在内存中不相邻，所以缓存命中性没有B+树好。

b+树的中间节点不保存数据，能容纳更多节点元素。

B树和B+树的共同优点

考虑磁盘IO的影响，它相对于内存来说是很慢的。数据库索引是存储在磁盘上的，当数据量大时，就不能把整个索引全部加载到内存了，只能逐一加载每一个磁盘页（对应索引树的节点）。所以我们要减少IO次数，对于树来说，IO次数就是树的高度，而“矮胖”就是b树的特征之一，m的大小取决于磁盘页的大小。

HashMap的实现，扩容机制，扩容时如何保证可操作?

扩容机制：

**1.7：**

当HashMap决定扩容时，会调用HashMap类中的resize(int newCapacity)方法，参数是新的table长度，一般为原长度的一倍且如果原有table长度已经达到了上限，就不再扩容，如果还未达到上限，则创建一个新的table，并调用transfer方法

    /\*\*

     \* Transfers all entries from current table to newTable.

     \*/

    void transfer(Entry[] newTable, boolean rehash) {

        int newCapacity = newTable.length;

        for (Entry<K,V> e : table) {

            while(null != e) {

                Entry<K,V> next = e.next;

                if (rehash) {

                    e.hash = null == e.key ? 0 : hash(e.key);

                }

                int i = indexFor(e.hash, newCapacity);

                e.next = newTable[i];

                newTable[i] = e;

                e = next;

            }

        }

}

transfer方法的作用是把原table的Node放到新的table中，使用的是头插法，也就是说，新table中链表的顺序和旧列表中是相反的，在HashMap线程不安全的情况下，这种头插法可能会导致环状节点

1.8：

    final Node<K,V>[] resize() {

        Node<K,V>[] oldTab = table;

        int oldCap = (oldTab == null) ? 0 : oldTab.length;

        int oldThr = threshold;

        int newCap, newThr = 0;

        if (oldCap > 0) {

            if (oldCap >= MAXIMUM\_CAPACITY) {

                threshold = Integer.MAX\_VALUE;

                return oldTab;

            }

            else if ((newCap = oldCap << 1) < MAXIMUM\_CAPACITY &&

                     oldCap >= DEFAULT\_INITIAL\_CAPACITY)                      //注释1

                newThr = oldThr << 1; // double threshold

        }

        else if (oldThr > 0) // initial capacity was placed in threshold

            newCap = oldThr;

        else {               // zero initial threshold signifies using defaults

            newCap = DEFAULT\_INITIAL\_CAPACITY;

            newThr = (int)(DEFAULT\_LOAD\_FACTOR \* DEFAULT\_INITIAL\_CAPACITY);

        }

        if (newThr == 0) {

            float ft = (float)newCap \* loadFactor;

            newThr = (newCap < MAXIMUM\_CAPACITY && ft < (float)MAXIMUM\_CAPACITY ?

                      (int)ft : Integer.MAX\_VALUE);

        }

        threshold = newThr;

        @SuppressWarnings({"rawtypes","unchecked"})

            Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];

        table = newTab;

        if (oldTab != null) {

            for (int j = 0; j < oldCap; ++j) {                                 //注释2

                Node<K,V> e;

                if ((e = oldTab[j]) != null) {

                    oldTab[j] = null;

                    if (e.next == null)                                        //注释3

                        newTab[e.hash & (newCap - 1)] = e;

                    else if (e instanceof TreeNode)

                        ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);

                    else { // preserve order

                        Node<K,V> loHead = null, loTail = null;

                        Node<K,V> hiHead = null, hiTail = null;

                        Node<K,V> next;

                        do {

                            next = e.next;

                            if ((e.hash & oldCap) == 0) {                      //注释4

                                if (loTail == null)                            //注释5

                                    loHead = e;

                                else

                                    loTail.next = e;                           //注释6

                                loTail = e;                                    //注释7

                            }

                            else {

                                if (hiTail == null)

                                    hiHead = e;

                                else

                                    hiTail.next = e;

                                hiTail = e;

                            }

                        } while ((e = next) != null);

                        if (loTail != null) {                                  /注释8

                            loTail.next = null;

                            newTab[j] = loHead;

                        }

                        if (hiTail != null) {

                            hiTail.next = null;

                            newTab[j + oldCap] = hiHead;

                        }

                    }

                }

            }

        }

        return newTab;

}

代码解析：

1，在resize()方法中，定义了oldCap参数，记录了原table的长度，定义了newCap参数，记录新table长度，newCap是oldCap长度的2倍（注释1），同时扩展点也乘2。

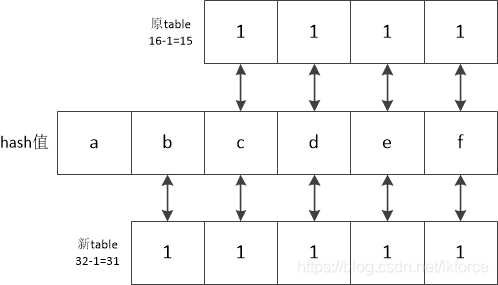
2，注释2是循环原table，把原table中的每个链表中的每个元素放入新table。

3，注释3，e.next==null，指的是链表中只有一个元素，所以直接把e放入新table，其中的e.hash & (newCap - 1)就是计算e在新table中的位置，和JDK1.7中的indexFor()方法是一回事。

4，注释// preserve order，这个注释是源码自带的，这里定义了4个变量：loHead，loTail，hiHead，hiTail，看起来可能有点眼晕，其实这里体现了JDK1.8对于计算节点在table中下标的新思路：

正常情况下，计算节点在table中的下标的方法是：hash&(oldTable.length-1)，扩容之后，table长度翻倍，计算table下标的方法是hash&(newTable.length-1)，也就是hash&(oldTable.length\*2-1)，于是我们有了这样的结论：这新旧两次计算下标的结果，要不然就相同，要不然就是新下标等于旧下标加上旧数组的长度。

举个例子，假设table原长度是16，扩容后长度32，那么一个hash值在扩容前后的table下标是这么计算的：



hash值的每个二进制位用abcde来表示，那么，hash和新旧table按位与的结果，最后4位显然是相同的，唯一可能出现的区别就在第5位，也就是hash值的b所在的那一位，如果b所在的那一位是0，那么新table按位与的结果和旧table的结果就相同，反之如果b所在的那一位是1，则新table按位与的结果就比旧table的结果多了10000（二进制），而这个二进制10000就是旧table的长度16。

换言之，hash值的新散列下标是不是需要加上旧table长度，只需要看看hash值第5位是不是1就行了，位运算的方法就是hash值和10000（也就是旧table长度）来按位与，其结果只可能是10000或者00000。

所以，注释4处的e.hash & oldCap，就是用于计算位置b到底是0还是1用的，只要其结果是0，则新散列下标就等于原散列下标，否则新散列坐标要在原散列坐标的基础上加上原table长度。

理解了上面的原理，这里的代码就好理解了，代码中定义的四个变量：

loHead，下标不变情况下的链表头

loTail，下标不变情况下的链表尾

hiHead，下标改变情况下的链表头

hiTail，下标改变情况下的链表尾

而注释4处的(e.hash & oldCap) == 0，就是代表散列下标不变的情况，这种情况下代码只使用了loHead和loTail两个参数，由他们组成了一个链表，否则将使用hiHead和hiTail参数。

其实e.hash & oldCap等于0和不等于0后的逻辑完全相同，只是用的变量不一样。

以等于0的情况为例，处理一个3-->5-->7的链表，过程如下：

首先处理节点3，e==3，e.next==5

1，注释5，一开始loTail是null，所以把3赋值给loHead。

2，注释7，把3赋值给loTail。

然后处理节点5，e==5，e.next==7

1，注释6，loTail有值，把e赋值给loTail.next，也就是3.next==5。

2，注释7，把5赋值给loTail。

现在新链表是3-->5，然后处理节点7，处理完之后，链表的顺序是3-->5-->7，loHead是3，loTail是7。可以看到，链表中节点顺序和原链表相同，不再是JDK1.7的倒序了。

代码到注释8这里就好理解了，

只要loTail不是null，说明链表中的元素在新table中的下标没变，所以新table的对应下标中放的是loHead，另外把loTail的next设为null

反之，hiTail不是null，说明链表中的元素在新table中的下标，应该是原下标加原table长度，新table对应下标处放的是hiHead，另外把hiTail的next设为null。

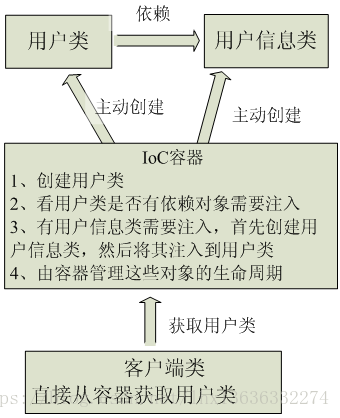
Redis扩容机制(渐进式单线程扩容)

即字典结构（底层为哈希表） 渐进式哈希

了解即可。。

<https://blog.csdn.net/belalds/article/details/93713491>

Spring Ioc的原理，如何实现，如何解决循环依赖?



IoC

对于spring框架来说，就是由spring来负责控制对象的生命周期和对象间的关系

Spring所倡导的开发方式就是如此，所有的类都会在spring容器中登记，告诉spring你是个什么东西，你需要什么东西，然后spring会在系统运行到适当的时候，把你要的东西主动给你，同时也把你交给其他需要你的东西。所有的类的创建、销毁都由 spring来控制，也就是说控制对象生存周期的不再是引用它的对象，而是spring。对于某个具体的对象而言，以前是它控制其他对象，现在是所有对象都被spring控制，所以这叫控制反转。

采用设计模式：抽象工厂模式

DI(反射实现注入):

IoC的一个重点是在系统运行中，动态的向某个对象提供它所需要的其他对象。这一点是通过DI（Dependency Injection，依赖注入）来实现的。比如对象A需要操作数据库，以前我们总是要在A中自己编写代码来获得一个Connection对象，有了 spring我们就只需要告诉spring，A中需要一个Connection，至于这个Connection怎么构造，何时构造，A不需要知道。在系统运行时，spring会在适当的时候制造一个Connection，然后像打针一样，注射到A当中，这样就完成了对各个对象之间关系的控制。A需要依赖 Connection才能正常运行，而这个Connection是由spring注入到A中的，依赖注入的名字就这么来的

Spring解决循环依赖：

Spring处理循环依赖的基本思路是这样的：

虽说要初始化一个Bean，必须要注入Bean里的依赖，才算初始化成功，但并不要求此时依赖的依赖也都注入成功，只要依赖对象的构造方法执行完了，这个依赖对象就算存在了，注入就算成功了，至于依赖的依赖，以后再初始化也来得及（参考Java的内存模型）。

因此，我们初始化一个Bean时，先调用Bean的构造方法，这个对象就在内存中存在了（对象里面的依赖还没有被注入），然后把这个对象保存下来，当循环依赖产生时，直接拿到之前保存的对象，于是循环依赖就被终止了，依赖注入也就顺利完成了。

eg：

假设对象A中有属性是对象B，对象B中也有属性是对象A，即A和B循环依赖。

创建对象A，调用A的构造，并把A保存下来。

然后准备注入对象A中的依赖，发现对象A依赖对象B，那么开始创建对象B。

调用B的构造，并把B保存下来。

然后准备注入B的构造，发现B依赖对象A，对象A之前已经创建了，直接获取A并把A注入B（注意此时的对象A还没有完全注入成功，对象A中的对象B还没有注入），于是B创建成功。

把创建成功的B注入A，于是A也创建成功了。

于是循环依赖就被解决了。

源码解析：

在注入一个对象的过程中，调用了这样一个方法：

Object sharedInstance = this.getSingleton(beanName);

这段代码在AbstractBeanFactory类的doGetBean()方法中。

这里得到的Object就是试图是要创建的对象，beanName就是要创建的对象的类名，这里getSingleton()方法的代码如下：

@Nullable

protected Object getSingleton(String beanName, boolean allowEarlyReference) {

    Object singletonObject = this.singletonObjects.get(beanName);

    if (singletonObject == null && isSingletonCurrentlyInCreation(beanName)) {

        synchronized (this.singletonObjects) {

            singletonObject = this.earlySingletonObjects.get(beanName);

            if (singletonObject == null && allowEarlyReference) {

                ObjectFactory<?> singletonFactory = this.singletonFactories.get(beanName);

                if (singletonFactory != null) {

                    singletonObject = singletonFactory.getObject();

                    this.earlySingletonObjects.put(beanName, singletonObject);

                    this.singletonFactories.remove(beanName);

                }

            }

        }

    }

    return singletonObject;

}

这个方法是Spring解决循环依赖的关键方法，在这个方法中，使用了三层列表来查询的方式，这三层列表分别是：

singletonObjects

earlySingletonObjects

singletonFactories

这个方法中用到的几个判断逻辑，体现了Spring解决循环依赖的思路，不过实际上对象被放入这三层的顺序是和方法查询的循序相反的，也就是说，在循环依赖出现时，对象往往会先进入singletonFactories，然后earlySingletonObjects，然后singletonObjects。

下面看一下这个方法的代码逻辑：

1，

Object singletonObject = this.singletonObjects.get(beanName);

方法首先从singletonObjects中获取对象，当Spring准备新建一个对象时，singletonObjects列表中是没有这个对象的，然后进入下一步。

2，

if (singletonObject == null && isSingletonCurrentlyInCreation(beanName))

除了判断null之外，有一个isSingletonCurrentlyInCreation的判断，实际上当Spring初始化了一个依赖注入的对象，但还没注入对象属性的时候，Spring会把这个bean加入singletonsCurrentlyInCreation这个set中，也就是把这个对象标记为正在创建的状态，这样，如果Spring发现要创建的bean在singletonObjects中没有，但在singletonsCurrentlyInCreation中有，基本上就可以认定为循环依赖了（在创建bean的过程中发现又要创建这个bean，说明bean的某个依赖又依赖了这个bean，即循环依赖）。

举个例子：对象A和对象B循环依赖，那么初始化对象A之后（执行了构造方法），要把A放入singletonsCurrentlyInCreation，对象A依赖了对象B，那么就要再初始化对象B，如果这个对象B又依赖了对象A，也就是形成了循环依赖，那么当我们注入对象B中的属性A时，进入这个代码逻辑，就会发现，我们要注入的对象A已经在singletonsCurrentlyInCreation中了，后面的逻辑就该处理这种循环依赖了。

3，

singletonObject = this.earlySingletonObjects.get(beanName);

这里引入了earlySingletonObjects列表，这是个为了循环依赖而存在的列表，从名字就可以看到，是个预创建的对象列表，刚刚创建的对象在这个列表里一般也没有。

4，

if (singletonObject == null && allowEarlyReference) {

                ObjectFactory<?> singletonFactory = this.singletonFactories.get(beanName);

earlySingletonObjects也没有则从singletonFactories中获取，前面说到singletonFactories是对象保存的第一步，实际上对象初始化后，可能还没有注入对象的依赖，就把对象放入了这个列表。

如果是循环依赖，此时的singletonFactories中一般是会存在目标对象的，举个例子：对象A和对象B循环依赖，那么初始化了对象A（执行了构造方法），还没有注入对象A的依赖时，就会把A放入singletonFactories，然后开始注入A的依赖，发现A依赖B，那么需要构对象B，构造过程也是执行了B的构造后就把B放到singletonFactories，然后开始注入B的依赖，发现B依赖A，在第二步中提到，此时A已经在singletonsCurrentlyInCreation列表里了，所以会进入此段代码逻辑，而且此时时对象A在singletonFactories中确实存在，因为这已经是第二次试图创建对象A了。

5，

if (singletonFactory != null) {

    singletonObject = singletonFactory.getObject();

    this.earlySingletonObjects.put(beanName, singletonObject);

    this.singletonFactories.remove(beanName);

}

代码到这里基本已经确定我们要创建的这个对象已经发生循环依赖了，然后Spring进行了这样的操作，把这个对象加入到earlySingletonObjects中，然后把该对象从singletonFactories中删掉。

6，其实上面5步已经执行完了该方法的代码，这里加的第6步是为了解释循环依赖的结果。在这个方法的代码之后，会把bean完整的进行初始化和依赖的注入，在完成了bean的初始化后，后面代码逻辑中会调用一个这样的方法：

getSingleton(String beanName, ObjectFactory<?> singletonFactory)

这个方法中有个小小的子方法addSingleton()，他的代码是这样的：

protected void addSingleton(String beanName, Object singletonObject) {

    synchronized (this.singletonObjects) {

        this.singletonObjects.put(beanName, singletonObject);

        this.singletonFactories.remove(beanName);

        this.earlySingletonObjects.remove(beanName);

        this.registeredSingletons.add(beanName);

    }

}

这个方法处理的是已经注入完依赖的bean，把bean放入singletonObjects中，并把bean从earlySingletonObjects和singletonFactories中删除，这个方法和上面分析的方法组成了Spring处理循环依赖的逻辑。

综上，Spring处理循环依赖的流程大概就是以下这样，假设对象A和对象B循环依赖





CAS概念、原子类实现原理：

Cas:

Compare and Swap, 翻译成比较并交换。 比较的是当前内存中存储的值与预期原值，交换的是新值与内存中的值。这个操作是硬件层面的指令，因此能够保证原子性。Java通过JNI（本地方法调用）来使用这个原子操作，也是乐观锁最常用的机制。

CAS操作包含三个操作数——内存位置、预期原值和新值。在执行CAS操作时，先进行Compare操作，即比较内存位置的值与预期原值是否相等，若相等，则执行Swap操作将新值放入该内存位置。若不相等，则不进行Swap操作

java.util.concurrent包中借助CAS实现了区别于synchronouse同步锁的一种乐观锁。

Eg:

如果把自增直接写成i++，那一定会出现并发问题，因为这不是原子操作，就不多说了。但是有了CAS操作之后，并发环境下的自增操作就可以很安全的实现了。下面来看一下如何借助CAS原子操作实现自增操作，先看一段伪代码。

            //自旋直到CAS操作成功

do{

oldValue = getCurrent(addr); //在执行CAS之前获取预期原值

newValue = oldValue + 1; //根据预期原值做增加操作的到新值

}while (!compareAndSwap(addr, oldValue, newValue)); //执行CAS操作

（1）首先获得预期原值，因为在自增情况下，新值是依赖于旧值的。

（2）通过计算得到新值。在这个过程中，可能有其他线程对该内存位置的值进行更新（自增），因为我们采用乐观锁的概念，并没有对变量进行加锁。

（3）再执行CAS操作。首先比较预期原值与当前内存位置的值是否相等。若相同，说明在这期间，没有其他线程对该变量进行更新，没有并发问题发生，则可以执行swap操作对旧值进行更新；若不同，则说明在这期间，有其他线程对变量进行了更新，当前的newValue其实是失效的，则要重新执行循环，即自旋，直至更新成功。

## 原子类AtomicInteger:

原子类的自增、加法等操作底层都是通过自旋CAS操作实现的，其核心原理就是我上面写的伪代码。

源码：

private static final Unsafe unsafe = Unsafe.getUnsafe();//最重要的Unsafe类，其中有对底层CAS操作的封装的方法

private static final long valueOffset;//value相对于对象在内存中的偏移量，在执行CAS操作时需要用到

static {//初始化求偏移量

try {

valueOffset = unsafe.objectFieldOffset

(AtomicInteger.class.getDeclaredField("value"));

} catch (Exception ex) { throw new Error(ex); }

}

public final int getAndIncrement() {//自增操作

return unsafe.getAndAddInt(this, valueOffset, 1);//调用Unsafe对象的方法

}

UnSafe类中的重要方法：

public final native boolean compareAndSwapObject(Object var1, long var2, Object var4, Object var5);

public final native boolean compareAndSwapInt(Object var1, long var2, int var4, int var5);

public final native boolean compareAndSwapLong(Object var1, long var2, long var4, long var6);

这些方法都是native的，是调用了底层操作系统的CAS指令。

**CAS缺点**

CAS虽然很高效的解决原子操作，但是CAS仍然存在三大问题。ABA问题，循环时间长开销大和只能保证一个共享变量的原子操作

1. ABA问题。因为CAS需要在操作值的时候检查下值有没有发生变化，如果没有发生变化则更新，但是如果一个值原来是A，变成了B，又变成了A，那么使用CAS进行检查时会发现它的值没有发生变化，但是实际上却变化了。ABA问题的解决思路就是使用版本号。在变量前面追加上版本号，每次变量更新的时候把版本号加一，那么A－B－A 就会变成1A-2B－3A。

从Java1.5开始JDK的atomic包里提供了一个类AtomicStampedReference来解决ABA问题。这个类的compareAndSet方法作用是首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。

关于ABA问题参考文档: http://blog.hesey.net/2011/09/resolve-aba-by-atomicstampedreference.html

2. 循环时间长开销大。自旋CAS如果长时间不成功，会给CPU带来非常大的执行开销。如果JVM能支持处理器提供的pause指令那么效率会有一定的提升，pause指令有两个作用，第一它可以延迟流水线执行指令（de-pipeline）,使CPU不会消耗过多的执行资源，延迟的时间取决于具体实现的版本，在一些处理器上延迟时间是零。第二它可以避免在退出循环的时候因内存顺序冲突（memory order violation）而引起CPU流水线被清空（CPU pipeline flush），从而提高CPU的执行效率。

3. 只能保证一个共享变量的原子操作。当对一个共享变量执行操作时，我们可以使用循环CAS的方式来保证原子操作，但是对多个共享变量操作时，循环CAS就无法保证操作的原子性，这个时候就可以用锁，或者有一个取巧的办法，就是把多个共享变量合并成一个共享变量来操作。比如有两个共享变量i＝2,j=a，合并一下ij=2a，然后用CAS来操作ij。从Java1.5开始JDK提供了AtomicReference类来保证引用对象之间的原子性，你可以把多个变量放在一个对象里来进行CAS操作。

## CAS与volatile

在Java的concurrent包中，有一种通用的实现方式，即CAS配合volatile来实现许多高并发类。

一般情况下实现流程：

（1）声明变量为volatile

（2）使用CAS条件更新来实现线程之间的同步。

（3）使用volatile变量的读/写和CAS所具有的volatile读和写的内存语义来实现线程之间的通信。

Java并发包的框架如图所示：



参考：https://blog.csdn.net/Hsuxu/article/details/9467651

TCP为什么是三次握手四次挥手?

参考：https://baijiahao.baidu.com/s?id=1654225744653405133很详细

数据库的隔离级别

四大特性ACID

**原子性（Atomicity）**: 原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚。失败回滚的操作事务，将不能对事物有任何影响

**一致性（Consistency）**: 一致性是指事务必须使数据库从一个一致性状态变换到另一个一致性状态，也就是说一个事务执行之前和执行之后都必须处于一致性状态。

数据库状态如何变化？每一次数据变更就会导致数据库的状态迁移。如果数据库的初始状态是C0，第一次事务T1的提交就会导致系统生成一个SYSTEM CHANGE NUMBER（SCN），这是数据库状态从C0转变成C1。执行第二个事务T2的时候数据库状态从c1变成c2，以此类推，执行第Tn次事务的时候数据库状态由C(n-1)变成Cn。

一致性可以从一致读和一致写两个方面来理解。

一致读 事务读取数据只能从一个状态中读取，不能从2个或者2个以上状态读取。也就是T(n)只能从C(n-1），C(n-2)... C(1)中的一个状态读取数据，不能一部分数据读取自C(n-1)，而另一部分数据读取自C(n-2)。

一致写 事务执行的数据变更只能基于上一个一致的状态，且只能体现在一个状态中。T(n)的变更结果只能基于C(n-1)，C(n-2), ...C(1)状态，且只能体现在C(n)状态中。也就是说，一个状态只能有一个事务变更数据，不允许有2个或者2个以上事务在一个状态中变更数据。至于具体一致写基于哪个状态，需要判断T(n)事务是否和T(n-1)，T(n-2),...T(1)有依赖关系

**隔离性（Isolation）:** **隔离性是指当多个用户并发访问数据库时，比如同时访问一张表，数据库每一个用户开启的事务，不能被其他事务所做的操作干扰，多个并发事务之间，应当相互隔离。**

**例如同时有T1和T2两个并发事务，从T1角度来看，T2要不在T1执行之前就已经结束，要么在T1执行完成后才开始。将多个事务隔离开，每个事务都不能访问到其他事务操作过程中的状态。**

**持久性（Durability）:** 持久性是指事务的操作，一旦提交，对于数据库中数据的改变是永久性的，即使数据库发生故障也不能丢失已提交事务所完成的改变。

没有隔离性下可能产生的问题：

**1. 脏读**

脏读是**指一个事务读取了未提交事务执行过程中的数据**。

当一个事务的操作正在多次修改数据，而在事务还未提交的时候，另外一个并发事务来读取了数据，就会导致读取到的数据并非是最终持久化之后的数据，这个数据就是脏读的数据。

最典型的例子就是银行转账，从A账户转账100到B账户，脚本命令为

update account set money = money + 100 where username = 'B';

update account set money = money - 100 where username = 'A';

在这个事务执行过程中，另外一个事务读取结果发现B账户中的钱已经到账，提示B钱已到账，B就进行了下一步的操作。但是最终转账事务失败，导致操作回滚。实际上B并未收到钱，但是进行了下一步的操作，造成了损失，这就是脏读。

**2. 不可重复读**

不可重复读是指对于**数据库中的某个数据，一个事务执行过程中多次查询返回不同查询结果，这就是在事务执行过程中，数据被其他事务提交修改了**。

不可重复读同脏读的区别在于，脏读是一个事务读取了另一未完成的事务执行过程中的数据，而不可重复读是一个事务执行过程中，另一事务提交并修改了当前事务正在读取的数据。

**3. 虚读(幻读)**

幻读是事务非独立执行时发生的一种现象，例如事务T1批量对一个表中某一列列值为1的数据修改为2的变更，但是在这时，事务T2对这张表插入了一条列值为1的数据，并完成提交。此时，如果事务T1查看刚刚完成操作的数据，发现还有一条列值为1的数据没有进行修改，而这条数据其实是T2刚刚提交插入的，这就是幻读。

**幻读和不可重复读都是读取了另一条已经提交的事务（这点同脏读不同），所不同的是不可重复读查询的都是同一个数据项，而幻读针对的是一批数据整体（比如数据的个数）**。

隔离级别：

**REPEATABLE READ Repeatable Read 可重复读**

MySQL数据库默认的隔离级别。该级别解决了READ UNCOMMITTED隔离级别导致的问题。它保证同一事务的多个实例在并发读取事务时，会“看到同样的”数据行。不过，这会导致另外一个棘手问题“幻读”。InnoDB和Falcon存储引擎通过多版本并发控制机制解决了幻读问题。

**READ COMMITTED Read Committed 读取提交内容**

大多数数据库系统的默认隔离级别（但是不是MySQL的默认隔离级别），满足了隔离的早先简单定义：一个事务开始时，只能“看见”已经提交事务所做的改变，一个事务从开始到提交前，所做的任何数据改变都是不可见的，除非已经提交。这种隔离级别也支持所谓的“不可重复读”。这意味着用户运行同一个语句两次，看到的结果是不同的。

**READ UNCOMMITTED Read UnCommitted 读取未提交内容**

在这个隔离级别，所有事务都可以“看到”未提交事务的执行结果。在这种级别上，可能会产生很多问题，除非用户真的知道自己在做什么，并有很好的理由选择这样做。本隔离级别很少用于实际应用，因为它的性能也不必其他性能好多少，而别的级别还有其他更多的优点。读取未提交数据，也被称为“脏读”

**SERIALIZABLE Serializable 可串行化**

该级别是最高级别的隔离级。它通过强制事务排序，使之不可能相互冲突，从而解决幻读问题。简而言之，SERIALIZABLE是在每个读的数据行上加锁。在这个级别，可能导致大量的超时Timeout和锁竞争Lock Contention现象，实际应用中很少使用到这个级别，但如果用户的应用为了数据的稳定性，需要强制减少并发的话，也可以选择这种隔离级

下面的表格总结了各种隔离级别和各自的缺点



修改隔离级别的方法：

全局修改需要修改MySql的全局文件mysql.ini，修改内容如下

1 #可选参数有：READ-UNCOMMITTED, READ-COMMITTED, REPEATABLE-READ, SERIALIZABLE.

2 [mysqld]

3 transaction-isolation = REPEATABLE-READ

**语句修改：**

查看：

mysql> select @@tx\_isolation;

+-----------------+

| @@tx\_isolation |

+-----------------+

| REPEATABLE-READ |

+-----------------+

1 row in set (0.00 sec)

**修改当前会话Session的隔离级:**

mysql> set session transaction isolation level read committed;

Query OK, 0 rows affected (0.00 sec)

mysql> select @@tx\_isolation;

+----------------+

| @@tx\_isolation |

+----------------+

| READ-COMMITTED |

+----------------+

1 row in set (0.00 sec)

**AutoCommit 事务自动提交：**

MySql中有AutoCommit参数，默认为on，也就是开启状态。它的作用是每一条单独的查询都是一个事务，自动开始，自动提交（语句执行完成就提交。如果你要适用select for update，而不手动调用 start transaction，这个for update的行锁机制等于没用，因为行锁在自动提交后就释放了）。所以事务隔离级别和锁机制即使你不显式调用start transaction，这种机制在单独一条语句查询中也是适用的。  
在命令行模式下可以使用下面的命令查看当前MySql的autocommit是否开启

**mysql> show variables like 'autocommit';**

**+---------------+-------+**

**| Variable\_name | Value |**

**+---------------+-------+**

**| autocommit | ON |**

**+---------------+-------+**

**1 row in set (0.00 sec)**

关闭autocommit：mysql> set autocommit=0;

0就是OFF，1就是ON。设置为OFF之后，则用户执行语句之后，将一直处于一个事务中，直到执行commit或者rollback，才会结束当前事务，重新开始新的事务。

锁相关：

共享锁

由读表操作加上的锁，加锁后其他用户只能获取该表或行的共享锁，不能获取排它锁，也就是说只能读不能写

排它锁

由写表操作加上的锁，加锁后其他用户不能获取该表或行的任何锁，典型是mysql事务

根据锁的范围，可以分为

表锁

给整张表加锁

行锁

给行数据加锁

因此锁可以分为表级共享锁、行级共享锁、表级排它锁、行级排它锁。

镜像二叉树.

http://blog.itpub.net/31561269/viewspace-2374664/ 详解

64匹马，8个赛道，找最快的4匹马

不计时，64匹马分成8组。为方便理解，前8次记录建立一个8x8矩阵，横坐标为组号，纵坐标为名次，第9次各组第一竞赛，将本次冠军（1，1）所在组放在第一列，并以此为矩阵排序。第10次取 （1，2）（1，3）（1，4） （2，1）（2，2）（2，3） （3，1）（3，2）竞赛 如果（1，2）进入前二 则取前三名与（1，1）为最快的四匹马 如果（1，2）未进入前二 则将（1，4）换成（4，1） 再进行第11轮竞赛，取前三名与（1，1）为最快的四匹马 综上所述，最快10轮，最慢11轮

给出两个升序数组A、B和长度m、n，求第k个大的

归并算法排序

给出数组A，长度为n，数组中元素的值位于[0,n - 1]之间,求是否有重复元素

从头到尾扫描数组每个数字，当扫描到下标为i的数字m时，首先比较m是不是等于i,如果是，继续扫描；如果不是，再拿m和第m个数字进行比较。如果他们相等，就找到第一个重复数字，如果不相等，交换两者位置。接下来重复上述过程，直到找到第一个重复数字

public static int findDuplicateNum(int nums[],int len){

if(nums==null||len<=0){

return -1;

}

for(int i=0;i<len;i++){ //存在负数或者大于n-1的数返回-1

if(nums[i]<0||nums[i]>len-1){

return -1;

}

}

int number=0;

for(int i=0;i<len;i++){

//判断下标为i的数字m和i是否相等

while(nums[i]!=i){//不相等，判断m和第m的个数是否相等

//相等

if(nums[i]==nums[nums[i]]){

number=nums[i];

return number;

}

//不相等，交换位置

int temp=nums[i];

nums[i]=nums[temp];

nums[temp]=temp;

}

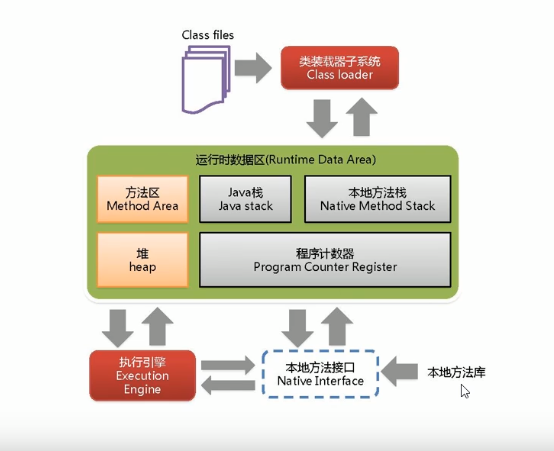
}

return -1;

}

JVM中什么时候会进行垃圾回收?什么样的对象是可以回收的?

1. jvm架构



类加载机制？

加载class文件到内存：

加载：导入class文件

验证：检查class文件的正确性

准备：为类的静态变量分配内存

解析：

初始化：对静态变量和静态代码初始化

类加载方式

显示加载类： class.forname

隐式加载类: new 对象的时候

类加载器

启动 类加载器（bootstrap） jre/lib/rt.jar

扩展 类加载器 (extension) jre/lib/ext/\*.jar

应用程序 类加载器（appclassloader）

自定义 类加载器 (继承java.lang.ClassLoader)

双亲委派机制？ 防止污染jkd源码 (虽然你写了 但我不加载)

当一个类收到了类加载请求时，不会自己先去加载这个类，而是将其委派给父类，由父类去加载，如果此时父类不能加载，反馈给子类，由子类去完成类的加载

naive 本地方法接口 本地方法库 本地方法栈

在本地方法栈中运行native方法 本地方法库作支持

实现jvm虚拟机 与 底层系统的交互

比如：thread.start() --->start0() java运行在jvm上，无法创建线程

pc寄存器？

每个线程都有自己的pc寄存器 标记线程执行到哪

方法区

存储类的模板信息（字段 构造方法 普通方法）

存储常量池

栈

运行方法 每执行一个方法就产生一个栈针 先进后出 栈顶部就是当前执行方法 执行完方法 就出栈

存储基本数据类型 对象引用 局部变量 实例方法

void A(){

A()

}

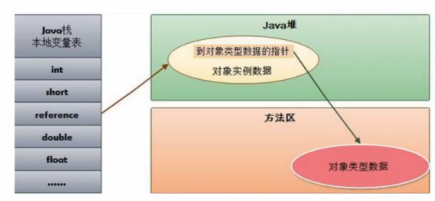
java.lang.stackOverFlowError

1. 栈 堆 方法区关系

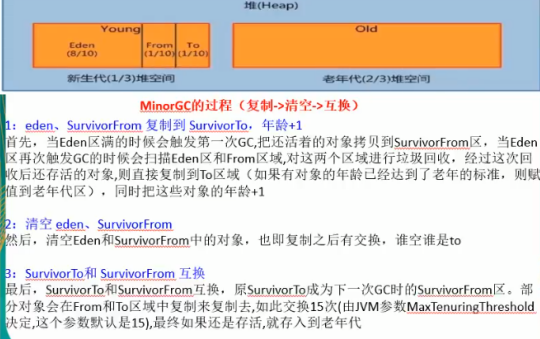
栈存储对象地址

堆存储方法区元数据的地址

方法区存储模板元数据



1. 堆



新生区(1/3)

伊甸区(1/3 的 8/10)

幸存0区（from 1/3 的 1/10）

幸存1区（to 1/3的1/10）

老年区(2/3)

新生对象进入eden区，当eden区满了，触发Minor GC（扫描eden区）,存活的对象移动到幸存from区,清空eden区

当eden区再次满了，触发Minor GC（扫描eden区和幸存from区），存活的对象复制到幸存to区，清空eden和from区

from 区变为to区 to区变为from区

当多次MinorGC仍然对象存在(默认15次交换)，进入老年区

当老年区满了，发生fullGC,

当实在清理不了老年区，OOM（堆内存溢出）

注意：对象一般是在eden区生死，一般扩大eden区调优

注意：gc会停止所有线程，尽量不要发生gc()

永久代存放JDK自带的Class元数据，不会被GC，关闭jvm才会被释放

1. 垃圾回收算法：system.gc（） 不会立即执行

引用计数法

多一次引用 计数器加一 少一次引用 计数器减一 减到0 就回收

缺点： 维护计数器消耗性能 难以处理循环引用

复制算法： 年轻代 整个from区都是垃圾 to区为空

发生YGC的时候 from区到to区

不会产生 内存碎片 （复制过程本身浪费空间）

标记清除：老年代，先标记 后清除 （有碎片）

标记整理：老年代，先标记，后压缩（没碎片）

1. JVM调优

| **参数** | **描述** |
| --- | --- |
| -Xms | 堆内存初始大小 |
| -Xmx（MaxHeapSize） | 堆内存最大允许大小 |
| -XX:PermSize | 非堆内存初始大小，一般应用设置初始化200m，最大1024m就够了 |
| -XX:MaxPermSize | 非堆内存最大允许大小 |
| -XX:NewSize（-Xns） | 年轻代内存初始大小 |
| -XX:MaxNewSize（-Xmn） | 年轻代内存最大允许大小，也可以缩写 |
| -XX:SurvivorRatio=8 | 年轻代中Eden区与Survivor区的容量比例值，默认为8，即8:1 |
| -XX:NewRatio=4 | 设置年轻的和老年代的内存比例为 1:4 |

Spring主要思想是什么?

Ioc（控制反转 抽象工厂模式） di（面型接口编程） aop（面向切面编程） 都是用来解耦的手段

详聊自由发挥。循环依赖 →三级缓存 ， 抽象工程模式，生命周期，spring原生扩展性

AQs有什么特点?

AQS原理

AQS：AbstractQuenedSynchronizer抽象的队列式同步器。是除了java自带的synchronized关键字之外的锁机制。

AQS的全称为（AbstractQueuedSynchronizer），这个类在java.util.concurrent.locks包

AQS的核心思想是，如果被请求的共享资源空闲，则将当前请求资源的线程设置为有效的工作线程，并将共享资源设置为锁定状态，如果被请求的共享资源被占用，那么就需要一套线程阻塞等待以及被唤醒时锁分配的机制，这个机制AQS是用CLH队列锁实现的，即将暂时获取不到锁的线程加入到队列中。

CLH（Craig，Landin，and Hagersten）队列是一个虚拟的双向队列，虚拟的双向队列即不存在队列实例，仅存在节点之间的关联关系。

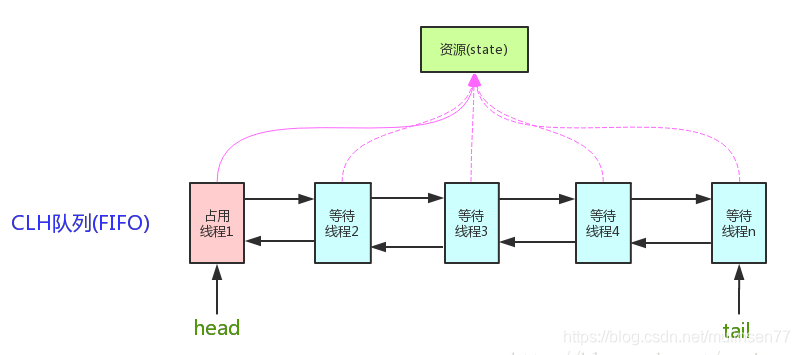
**AQS是将每一条请求共享资源的线程封装成一个CLH锁队列的一个结点（Node）**，来实现锁的分配。

**AQS就是基于CLH队列，用volatile修饰共享变量state，线程通过CAS去改变状态符，成功则获取锁成功，失败则进入等待队列，等待被唤醒**。

\*\*注意：AQS是自旋锁：\*\*在等待唤醒的时候，经常会使用自旋（while(!cas())）的方式，不停地尝试获取锁，直到被其他线程获取成功

**实现了AQS的锁有：自旋锁、互斥锁、读锁写锁、条件产量、信号量、栅栏都是AQS的衍生物**

AQS实现的具体方式如下：



如图示，AQS维护了一个volatile int state和一个FIFO线程等待队列，多线程争用资源被阻塞的时候就会进入这个队列。state就是共享资源，其访问方式有如下三种：

getState();setState();compareAndSetState();

AQS 定义了两种资源共享方式：

**1.Exclusive**：独占，只有一个线程能执行，如ReentrantLock

**2.Share**：共享，多个线程可以同时执行，如Semaphore、CountDownLatch、ReadWriteLock，CyclicBarrier

不同的自定义的同步器争用共享资源的方式也不同。

AQS底层使用了模板方法模式

同步器的设计是基于模板方法模式的，如果需要自定义同步器一般的方式是这样（模板方法模式很经典的一个应用）：

使用者继承AbstractQueuedSynchronizer并重写指定的方法。（这些重写方法很简单，无非是对于共享资源state的获取和释放）

将AQS组合在自定义同步组件的实现中，并调用其模板方法，而这些模板方法会调用使用者重写的方法。

这和我们以往通过实现接口的方式有很大区别，这是模板方法模式很经典的一个运用。

自定义同步器在实现的时候只需要实现共享资源state的获取和释放方式即可，至于具体线程等待队列的维护，AQS已经在顶层实现好了。自定义同步器实现的时候主要实现下面几种方法：

isHeldExclusively()：该线程是否正在独占资源。只有用到condition才需要去实现它。

tryAcquire(int)：独占方式。尝试获取资源，成功则返回true，失败则返回false。

tryRelease(int)：独占方式。尝试释放资源，成功则返回true，失败则返回false。

tryAcquireShared(int)：共享方式。尝试获取资源。负数表示失败；0表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。

tryReleaseShared(int)：共享方式。尝试释放资源，如果释放后允许唤醒后续等待结点返回true，否则返回false。

**ReentrantLock**为例，（可重入独占式锁）：state初始化为0，表示未锁定状态，A线程lock()时，会调用tryAcquire()独占锁并将state+1.之后其他线程再想tryAcquire的时候就会失败，直到A线程unlock（）到state=0为止，其他线程才有机会获取该锁。A释放锁之前，自己也是可以重复获取此锁（state累加），这就是可重入的概念。

注意：获取多少次锁就要释放多少次锁，保证state是能回到零态的。

以CountDownLatch为例，任务分N个子线程去执行，state就初始化 为N，N个线程并行执行，每个线程执行完之后countDown（）一次，state就会CAS减一。当N子线程全部执行完毕，state=0，会unpark()主调用线程，主调用线程就会从await()函数返回，继续之后的动作。

一般来说，自定义同步器要么是独占方法，要么是共享方式，他们也只需实现tryAcquire-tryRelease、tryAcquireShared-tryReleaseShared中的一种即可。但AQS也支持自定义同步器同时实现独占和共享两种方式，如ReentrantReadWriteLock。

　在acquire() acquireShared()两种方式下，线程在等待队列中都是忽略中断的，acquireInterruptibly()/acquireSharedInterruptibly()是支持响应中断的。

AQS的简单应用

Mutex：不可重入互斥锁，锁资源（state）只有两种状态：0：未被锁定；1：锁定。

class Mutex implements Lock, java.io.Serializable {

// 自定义同步器

private static class Sync extends AbstractQueuedSynchronizer {

// 判断是否锁定状态

protected boolean isHeldExclusively() {

return getState() == 1;

}

// 尝试获取资源，立即返回。成功则返回true，否则false。

public boolean tryAcquire(int acquires) {

assert acquires == 1; // 这里限定只能为1个量

if (compareAndSetState(0, 1)) {//state为0才设置为1，不可重入！

setExclusiveOwnerThread(Thread.currentThread());//设置为当前线程独占资源

return true;

}

return false;

}

// 尝试释放资源，立即返回。成功则为true，否则false。

protected boolean tryRelease(int releases) {

assert releases == 1; // 限定为1个量

if (getState() == 0)//既然来释放，那肯定就是已占有状态了。只是为了保险，多层判断！

throw new IllegalMonitorStateException();

setExclusiveOwnerThread(null);

setState(0);//释放资源，放弃占有状态

return true;

}

}

// 真正同步类的实现都依赖继承于AQS的自定义同步器！

private final Sync sync = new Sync();

//lock<-->acquire。两者语义一样：获取资源，即便等待，直到成功才返回。

public void lock() {

sync.acquire(1);

}

//tryLock<-->tryAcquire。两者语义一样：尝试获取资源，要求立即返回。成功则为true，失败则为false。

public boolean tryLock() {

return sync.tryAcquire(1);

}

//unlock<-->release。两者语文一样：释放资源。

public void unlock() {

sync.release(1);

}

//锁是否占有状态

public boolean isLocked() {

return sync.isHeldExclusively();

}

}

转载：

<https://blog.csdn.net/mulinsen77/article/details/84583716>

详解：

<https://blog.csdn.net/qq_50695280/article/details/115099872>

各种网络协议

DNs在网络层用哪个协议，为什么

DNS占用53号端口，同时使用TCP和UDP协议。那么DNS在什么情况下使用这两种协议？

**DNS在区域传输的时候使用TCP协议，其他时候使用UDP协议。**

**DNS区域传输的时候使用TCP协议**：

1.辅域名服务器会定时（一般3小时）向主域名服务器进行查询以便了解数据是否有变动。如有变动，会执行一次区域传送，进行数据同步。区域传送使用TCP而不是UDP，因为数据同步传送的数据量比一个请求应答的数据量要多得多。

2.TCP是一种可靠连接，保证了数据的准确性。

**域名解析时使用UDP协议：**

客户端向DNS服务器查询域名，一般返回的内容都不超过512字节，用UDP传输即可。不用经过三次握手，这样DNS服务器负载更低，响应更快。理论上说，客户端也可以指定向DNS服务器查询时用TCP，但事实上，很多DNS服务器进行配置的时候，仅支持UDP查询包。

TCP与UDP的区别：

TCP与UDP：TCP和UDP都是位于OSI模型中的传输层中。

TCP是一种面向连接的协议，提供可靠的数据传输，一般服务质量要求比较高的情况，使用这个协议。、

UDP---用户数据报协议，是一种无连接的传输层协议，提供面向事务的简单不可靠信息传送服务。

UDP和TCP协议的主要区别是两者在如何实现信息的可靠传递方面不同。TCP协议中包含了专门的传递保证机制，当数据接收方收到发送方传来的信息时，会自动向发送方发出确认消息；发送方只有在接收到该确认消息之后才继续传送其它信息，否则将一直等待直到收到确认信息为止。 与TCP不同，UDP协议并不提供数据传送的保证机制。如果在从发送方到接收方的传递过程中出现数据报的丢失，协议本身并不能做出任何检测或提示。因此，通常人们把UDP协议称为不可靠的传输协议。相对于TCP协议，UDP协议的另外一个不同之处在于如何接收突发性的多个数据报。不同于TCP，UDP并不能确保数据的发送和接收顺序。事实上，UDP协议的这种乱序性基本上很少出现，通常只会在网络非常拥挤的情况下才有可能发生。

既然UDP是一种不可靠的网络协议，那么还有什么使用价值或必要呢？其实不然，在有些情况下UDP协议可能会变得非常有用。因为UDP具有TCP所望尘莫及的速度优势。虽然TCP协议中植入了各种安全保障功能，但是在实际执行的过程中会占用大量的系统开销，无疑使速度受到严重的影响。反观UDP由于排除了信息可靠传递机制，将安全和排序等功能移交给上层应用来完成，极大降低了执行时间，使速度得到了保证。

总结：

**TCP基于面向连接的协议，数据传输可靠，传输速度慢，适用于传输大量数据，可靠性要求高的场合。**

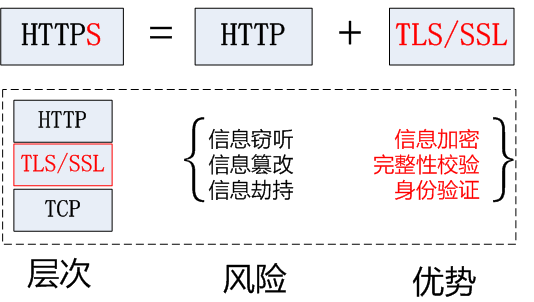
**UDP协议面向非连接协议，数据传输不可靠，传输速度快，适用于一次只传送少量数据、对可靠性要求不高的应用环境。**

介绍HTTPs协议，详述ssL建立连接过程

HTTPS协议详解

HTTPS (Secure Hypertext Transfer Protocol)安全超文本传输协议，是一个安全通信通道，它基于HTTP开发用于在客户计算机和服务器之间交换信息。它使用安全套接字层(SSL)进行信息交换，简单来说它是HTTP的安全版,是使用TLS/SSL加密的HTTP协议。HTTP协议采用明文传输信息，存在信息窃听、信息篡改和信息劫持的风险，而协议TLS/SSL具有身份验证、信息加密和完整性校验的功能，可以避免此类问题发生。

TLS/SSL全称安全传输层协议Transport Layer Security, 是介于TCP和HTTP之间的一层安全协议，不影响原有的TCP协议和HTTP协议，所以使用HTTPS基本上不需要对HTTP页面进行太多的改造



SSL工作流程：

**服务器认证阶段：**

客户端向服务器发送一个开始信息“Hello”以便开始一个新的会话连接；

服务器根据客户的信息确定是否需要生成新的主密钥，如需要则服务器在响应客户的“Hello”信息时将包含生成主密钥所需的信息；

客户根据收到的服务器响应信息，产生一个主密钥，并用服务器的公开密钥加密后传给服务器；

服务器回复该主密钥，并返回给客户一个用主密钥认证的信息，以此让客户认证服务器。

用户认证阶段：在此之前，服务器已经通过了客户认证，这一阶段主要完成对客户的认证。经认证的服务器发送一个提问给客户，客户则返回（数字）签名后的提问和其公开密钥，从而向服务器提供认证。

SSL协议提供的安全通道有以下三个特性：

机密性：SSL协议使用密钥加密通信数据。

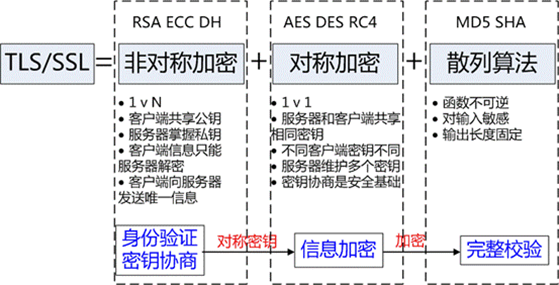
可靠性：服务器和客户都会被认证，客户的认证是可选的。

完整性：SSL协议会对传送的数据进行完整性检查。

TLS/SSL工作原理

HTTPS协议的主要功能基本都依赖于TLS/SSL协议，本节分析TLS/SSL协议工作原理。

TLS/SSL的功能实现主要依赖于三类基本算法：**散列函数 Hash、对称加密和非对称加密**，其利用非对称加密实现身份认证和密钥协商，对称加密算法采用协商的密钥对数据加密，基于散列函数验证信息的完整性。



#### 三种算法介绍

**散列函数Hash**

常见的有 MD5、SHA1、SHA256，该类函数特点是**函数单向不可逆、对输入非常敏感、输出长度固定**，针对数据的任何修改都会改变散列函数的结果，用于防止信息篡改并验证数据的完整性;

在信息传输过程中，散列函数不能单独实现信息防篡改，因为明文传输，中间人可以修改信息之后重新计算信息摘要，因此需要对传输的信息以及信息摘要进行加密;

**对称加密**

常见的有 AES-CBC、DES、3DES、AES-GCM等，相同的密钥可以用于信息的加密和解密，掌握密钥才能获取信息，能够防止信息窃听，**通信方式是1对1;**

对称加密的优势是信息传输1对1，需要共享相同的密码，密码的安全是保证信息安全的基础，服务器和 N 个客户端通信，需要维持 N 个密码记录，且缺少修改密码的机制;

**非对称加密**

即常见的 RSA 算法，还包括 ECC、DH 等算法，算法特点是，**密钥成对出现**，**一般称为公钥(公开)和私钥(保密)**，公钥加密的信息只能私钥解开，私钥加密的信息只能公钥解开。因此掌握公钥的不同客户端之间不能互相解密信息，只能和掌握私钥的服务器进行加密通信，**服务器可以实现1对多的通信**，客户端也可以用来验证掌握私钥的服务器身份。

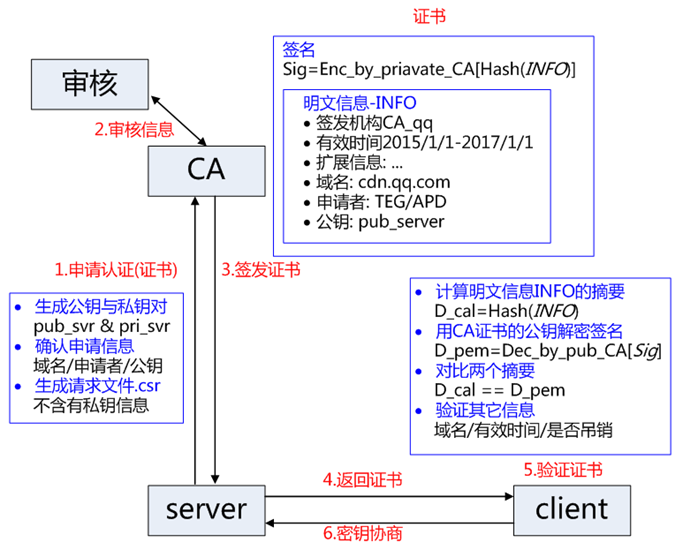
非对称加密的特点是信息传输1对多，服务器只需要维持一个私钥就能够和多个客户端进行加密通信，但服务器发出的信息能够被所有的客户端解密，且该**算法的计算复杂，加密速度慢。**

**总结**

结合三类算法的特点，TLS的基本工作方式是，**客户端使用非对称加密与服务器进行通信，实现身份验证并协商对称加密使用的密钥，然后对称加密算法采用协商密钥对信息以及信息摘要进行加密通信，不同的节点之间采用的对称密钥不同，从而可以保证信息只能通信双方获取**

身份验证CA和证书

CA负责审核信息，然后对关键信息利用私钥进行"签名"，公开对应的公钥，客户端可以利用公钥验证签名。CA也可以吊销已经签发的证书，基本的方式包括两类 CRL 文件和  
OCSP。CA使用具体的流程如下：



详情见：<https://www.cnblogs.com/evan-blog/p/9867561.html>

HTTP与HTTPS的区别

1. HTTPS是加密传输协议，HTTP是名文传输协议;
2. HTTPS需要用到SSL证书，而HTTP不用;
3. HTTPS比HTTP更加安全，对搜索引擎更友好，利于SEO【参考：（1）为保护用户隐私安全,谷歌优先索引HTTPS网页、（2）百度开放收录https站点，https全网化势不可挡】;
4. HTTPS标准端口443，HTTP标准端口80;
5. HTTPS基于传输层，HTTP基于应用层;
6. HTTPS在浏览器显示绿色安全锁，HTTP没有显示;

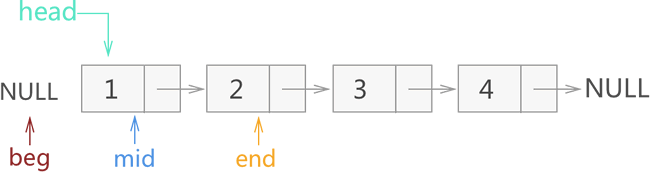


代码:反转单链表，复杂链表复制

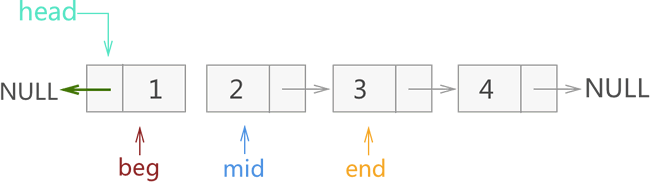
反转单链表：

**迭代反转链表**：从当前链表的首元节点开始，一直遍历至链表的最后一个节点，这期间会逐个改变所遍历到的节点的指针域，另其指向前一个节点

实现方法也很简单，借助 3 个指针即可。以图 1 中建立的链表为例，首先我们定义 3 个指针并分别命名为 beg、mid、end。

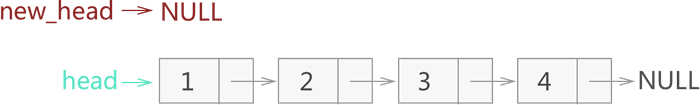


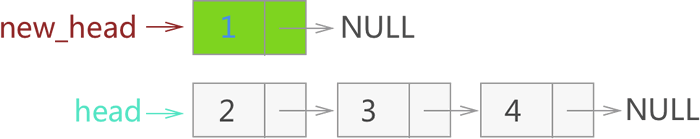
遍历链表的过程就等价为：3 个指针每次各向后移动一个节点，直至 mid 指向链表中最后一个节点（此时 end 为 NULL ）。需要注意的是，这 3 个指针每移动之前，都需要做一步操作，即改变 mid 所指节点的指针域，另其指向和 beg 相同。

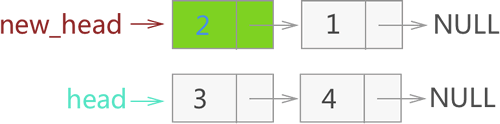


**头插法反转链表：**

在原有链表的基础上，依次将位于链表头部的节点摘下，然后采用从头部插入的方式生成一个新链表，则此链表即为原链表的反转版。







代码：

**public** **static** **class** **Node** {

**public** **int** **value**;

**public** Node next;

**public** **Node**(**int** data) {

**this**.**value** = data;

}

}

**public** **static** Node **reverseList**(Node node) {

Node pre = null;

Node next = null;

**while** (node != null) {

next = node.next;

node.next = pre;

pre = node;

node = next;

}

**return** pre;

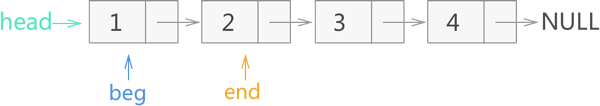
}

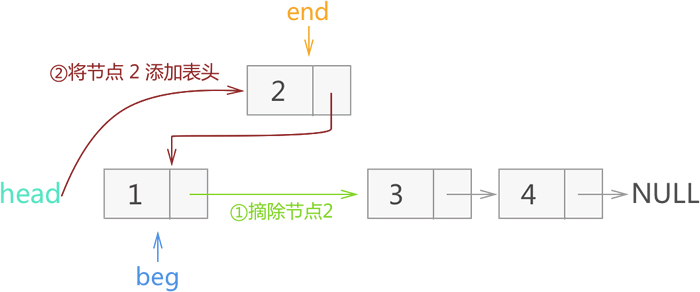
解析：准备两个空结点 pre用来保存先前结点、next用来做临时变量

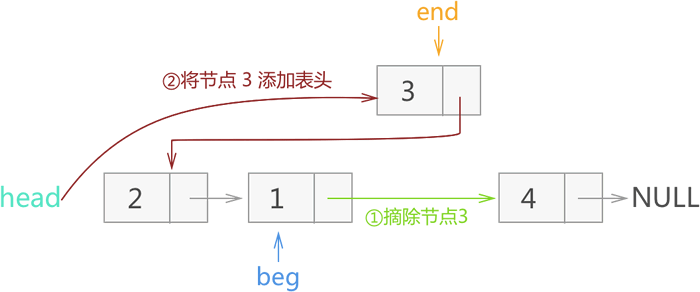
* 在头结点node遍历的时候此时为1结点
  + next = 1结点.next(2结点)
  + 1结点.next=pre(null)
  + pre = 1结点
  + node = 2结点
* 进行下一次循环node=2结点
  + next = 2结点.next(3结点)
  + 2结点.next=pre(1结点)=>即完成2->1
  + pre = 2结点
  + node = 3结点
* 进行循环...

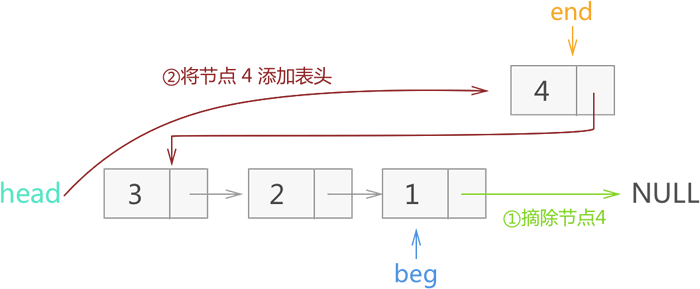
**就地逆置法反转链表**：

和头插法的实现思想类似，唯一的区别在于，头插法是通过建立一个新链表实现的，而就地逆置法则是直接对原链表做修改，从而实现将原链表反转









复杂链表复制：

思路一

1.创建一个map 遍历链表将每个结点 和复制当前结点的结点 构成一个key 和value放进map中

2.再次遍历链表 将每个结点作为key去查找map中的val 即复制的结点 它的next域就是 当前结点

的next在map中的val 他的random域就是 当前结点的random在map中的val

这种方法的时间复杂度为0(n) 额外空间复杂度为O(n)

下面介绍一种时间复杂度为0(n) 额外空间复杂度为O(1)的方法

思路二

利用和思路一同样的思想 只需要将复制的节点放在当前的结点的后面 则复制结点的random域就是当前节点的random域下一个结点 最后重新拆分链表就是了

\*\*请看代码结合注释理解 **拆分链表部分好好看看** \*\*

public class Solution {

public RandomListNode Clone(RandomListNode pHead) {

if(pHead == null) {

return null;

}

RandomListNode node = pHead;

// 遍历链表将复制结点放在当前结点的后面

while(node != null) {

RandomListNode newNode = new RandomListNode(node.label);

newNode.next = node.next;

node.next = newNode;

node = newNode.next;

}

node = pHead;

// 遍历链表 将复制结点的random指向当前结点的random的next

while(node != null && node.next != null) {

if(node.random != null) {

node.next.random = node.random.next;

}

node = node.next.next;

}

RandomListNode res = pHead.next;

node = pHead;

RandomListNode node1 = res;

// 拆分链表

while(node != null && node.next != null) {

node.next = node.next.next;

node1.next = node.next == null ? null : node.next.next;

node1 = node1.next;

node = node.next;

}

return res;

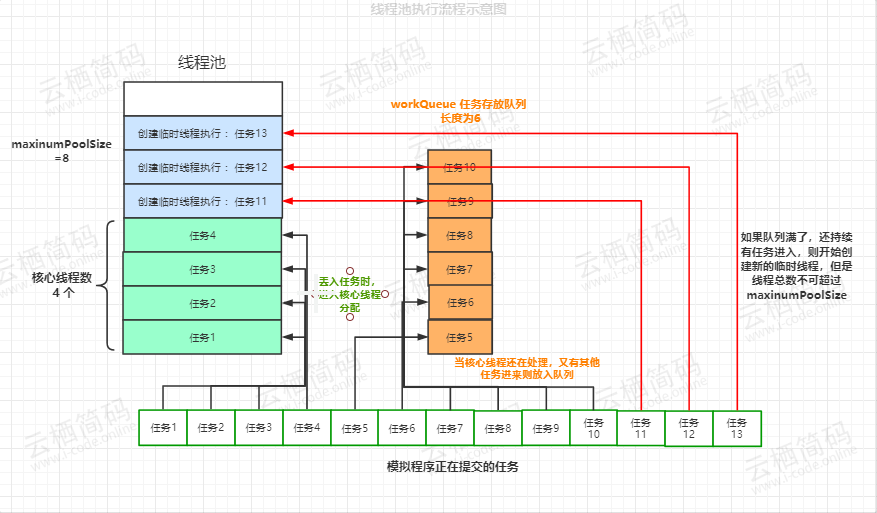
}

}

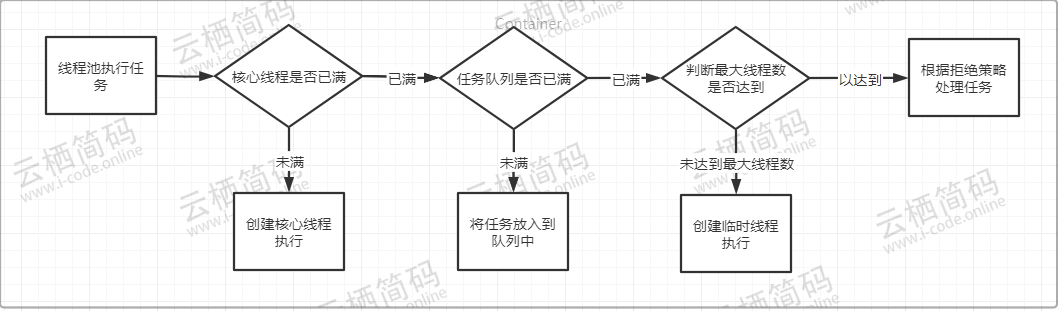
线程池参数的含义:



* 面展示的参数中第三个和第四个参数算是一个，时间和单位的组合
* corePoolSize 代表核心线程数，也就是正常情况下创建工作的线程数，这些线程创建后并不会消除，而是一种常驻线程
* maxinumPoolSize 代表的是最大线程数，它与核心线程数相对应，表示最大允许被创建的线程数，比如当前任务较多，将核心线程数都用完了，还无法满足需求时，此时就会创建新的线程，但是线程池内线程总数不会超过最大线程数



* keepAliveTime、unit 表示超出核心线程数之外的线程的空闲存活时间，也就是核心线程不会消除，但是超出核心线程数的部分线程如果空闲一定的时间则会被消除,我们可以通过 setKeepAliveTime 来设置空闲时间
* workQueue 用来存放待执行的任务，假设我们现在核心线程都已被使用，还有任务进来则全部放入队列，直到整个队列被放满但任务还再持续进入则会开始创建新的线程
* ThreadFactory 实际上是一个线程工厂，用来生产线程执行任务。我们可以选择使用默认的创建工厂，产生的线程都在同一个组内，拥有相同的优先级，且都不是守护线程。当然我们也可以选择自定义线程工厂，一般我们会根据业务来制定不同的线程工厂
* Handler 任务拒绝策略，有两种情况，第一种是当我们调用shutdown 等方法关闭线程池后，这时候即使线程池内部还有没执行完的任务正在执行，但是由于线程池已经关闭，我们再继续想线程池提交任务就会遭到拒绝。另一种情况就是当达到最大线程数，线程池已经没有能力继续处理新提交的任务时，这是也就拒绝



Innodb的索引实现

　InnoDB使用的是聚簇索引，将主键组织到一棵B+树中，而行数据就储存在叶子节点上，若使用"where id = 14"这样的条件查找主键，则按照B+树的检索算法即可查找到对应的叶节点，之后获得行数据。若对Name列进行条件搜索，则需要两个步骤：第一步在辅助索引B+树中检索Name，到达其叶子节点获取对应的主键。第二步使用主键在主索引B+树种再执行一次B+树检索操作，最终到达叶子节点即可获取整行数据。

聚簇索引按照如下规则创建：

当定义了主键后，InnoDB会利用主键来生成其聚簇索引；

如果没有主键，InnoDB会选择一个非空的唯一索引来创建聚簇索引；

如果这也没有，InnoDB会隐式的创建一个自增的列(rowid)来作为聚簇索引。

除了主键索引之外的索引，成为二级索引（Secondary Index）。二级索引可以有多个，二级索引建立在经常查询的列上。与聚簇索引的区别在于二级索引的叶子节点中存放的是除了这几个列外用来回表的主键信息（指针）。

所为回表：就是在使用二级索引时，因为二级索引只存储了部分数据，如果根据键值查找的数据不能包含全部目标数据，就需要根据二级索引的键值的主键信息，去聚簇索引的全部数据。然后根据完整数据取出所需要的列。这种在二级索引不能找到全部列的现象称为“非索引覆盖”，需要两次B+树查询，反之称为索引覆盖。所以索引需要平衡考虑，多建索引有利于查询，但是占用空间大还影响写入性能。即索引要精有用。

详解及优化：<https://blog.csdn.net/bohu83/article/details/81104432>

　　MyISM使用的是非聚簇索引，非聚簇索引的两棵B+树看上去没什么不同，节点的结构完全一致只是存储的内容不同而已，主键索引B+树的节点存储了主键，辅助键索引B+树存储了辅助键。表数据存储在独立的地方，这两颗B+树的叶子节点都使用一个地址指向真正的表数据，对于表数据来说，这两个键没有任何差别。由于索引树是独立的，通过辅助键检索无需访问主键的索引树。

volatile关键字的作用

volatile关键字是由JVM提供的最轻量级同步机制。与被滥用的synchronized不同，我们并不习惯使用它。想要正确且完全的理解它并不容易

修饰变量 在指令前后都加一层内存屏障保证指令顺序

保证可见性

不保证原子性

禁止指令重排

详解：需理解java内存机制 见前

<https://blog.csdn.net/u012723673/article/details/80682208>

乐**观锁、悲观锁**

详解：需要一定理解

https://www.jianshu.com/p/d2ac26ca6525

**Redis的使用，分布式锁的实现**

可靠性

首先，为了确保分布式锁可用，我们至少要确保锁的实现同时满足以下四个条件：

1. 互斥性。在任意时刻，只有一个客户端能持有锁。
2. 不会发生死锁。即使有一个客户端在持有锁的期间崩溃而没有主动解锁，也能保证后续其他客户端能加锁。
3. 具有容错性。只要大部分的Redis节点正常运行，客户端就可以加锁和解锁。
4. 解铃还须系铃人。加锁和解锁必须是同一个客户端，客户端自己不能把别人加的锁给解了。

组件依赖

首先我们要通过Maven引入Jedis开源组件，在pom.xml文件加入下面的代码：

<dependency>

<groupId>redis.clients</groupId>

<artifactId>jedis</artifactId>

<version>2.9.0</version>

</dependency>

代码实现

加锁：

public class RedisTool {

private static final String LOCK\_SUCCESS = "OK";

private static final String SET\_IF\_NOT\_EXIST = "NX";

private static final String SET\_WITH\_EXPIRE\_TIME = "PX";

/\*\*

\* 尝试获取分布式锁

\* @param jedis Redis客户端

\* @param lockKey 锁

\* @param requestId 请求标识

\* @param expireTime 超期时间

\* @return 是否获取成功

\*/

public static boolean tryGetDistributedLock(Jedis jedis, String lockKey, String requestId, int expireTime) {

String result = jedis.set(lockKey, requestId, SET\_IF\_NOT\_EXIST, SET\_WITH\_EXPIRE\_TIME, expireTime);

if (LOCK\_SUCCESS.equals(result)) {

return true;

}

return false;

}

}

可以看到，我们加锁就一行代码：jedis.set(String key, String value, String nxxx, String expx, int time)，这个set()方法一共有五个形参：

* 第一个为key，我们使用key来当锁，因为key是唯一的。
* 第二个为value，我们传的是requestId，很多童鞋可能不明白，有key作为锁不就够了吗，为什么还要用到value？原因就是我们在上面讲到可靠性时，分布式锁要满足第四个条件解铃还须系铃人，通过给value赋值为requestId，我们就知道这把锁是哪个请求加的了，在解锁的时候就可以有依据。requestId可以使用UUID.randomUUID().toString()方法生成。
* 第三个为nxxx，这个参数我们填的是NX，意思是SET IF NOT EXIST，即当key不存在时，我们进行set操作；若key已经存在，则不做任何操作；
* 第四个为expx，这个参数我们传的是PX，意思是我们要给这个key加一个过期的设置，具体时间由第五个参数决定。
* 第五个为time，与第四个参数相呼应，代表key的过期时间。

总的来说，执行上面的set()方法就只会导致两种结果：1. 当前没有锁（key不存在），那么就进行加锁操作，并对锁设置个有效期，同时value表示加锁的客户端。2. 已有锁存在，不做任何操作。

心细的童鞋就会发现了，我们的加锁代码满足我们可靠性里描述的三个条件。首先，set()加入了NX参数，可以保证如果已有key存在，则函数不会调用成功，也就是只有一个客户端能持有锁，满足互斥性。其次，由于我们对锁设置了过期时间，即使锁的持有者后续发生崩溃而没有解锁，锁也会因为到了过期时间而自动解锁（即key被删除），不会发生死锁。最后，因为我们将value赋值为requestId，代表加锁的客户端请求标识，那么在客户端在解锁的时候就可以进行校验是否是同一个客户端。由于我们只考虑Redis单机部署的场景，所以容错性我们暂不考虑。

解锁：

public class RedisTool {

private static final Long RELEASE\_SUCCESS = 1L;

/\*\*

\* 释放分布式锁

\* @param jedis Redis客户端

\* @param lockKey 锁

\* @param requestId 请求标识

\* @return 是否释放成功

\*/

public static boolean releaseDistributedLock(Jedis jedis, String lockKey, String requestId) {

String script = "if redis.call('get', KEYS[1]) == ARGV[1] then return redis.call('del', KEYS[1]) else return 0 end";

Object result = jedis.eval(script, Collections.singletonList(lockKey), Collections.singletonList(requestId));

if (RELEASE\_SUCCESS.equals(result)) {

return true;

}

return false;

}

}

第一行代码，我们写了一个简单的Lua脚本代码，上一次见到这个编程语言还是在《黑客与画家》里，没想到这次居然用上了。第二行代码，我们将Lua代码传到jedis.eval()方法里，并使参数KEYS[1]赋值为lockKey，ARGV[1]赋值为requestId。eval()方法是将Lua代码交给Redis服务端执行。

那么这段Lua代码的功能是什么呢？其实很简单，首先获取锁对应的value值，检查是否与requestId相等，如果相等则删除锁（解锁）。那么为什么要使用Lua语言来实现呢？因为要确保上述操作是原子性的。关于非原子性会带来什么问题，可以阅读[【解锁代码-错误示例2】](http://wudashan.cn/2017/10/23/Redis-Distributed-Lock-Implement/#releaseLock-wrongDemo2) 。那么为什么执行eval()方法可以确保原子性，源于Redis的特性，下面是官网对eval命令的部分解释：

简单来说，就是在eval命令执行Lua代码的时候，Lua代码将被当成一个命令去执行，并且直到eval命令执行完成，Redis才会执行其他命令。

详解：<https://www.kancloud.cn/cfun_good/knowledge/1367275>

用过哪些设计模式?

自由发挥。。。